A Variational Formulation of Dissipative Quasicontinuum Methods: Implementation Guide

Ondřej Rokoš[a,*], Lars Beex[b], Jan Zeman[a], Ron Peerlings[c]

[a]*Department of Mechanics, Faculty of Civil Engineering, Czech Technical University in Prague, Thákurova 7, 166 29 Prague 6, Czech Republic.*
[b]*Faculté des Sciences, de la Technologie et de la Communication, Campus Kirchberg, Université du Luxembourg 6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg.*
[c]*Department of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.*

Abstract

The purpose of this guide is to provide a description of the MATLAB® implementation accompanying the paper Rokoš et al., A Variational Formulation of Dissipative Quasicontinuum Methods, *Int. J. Solids Struct.*, 102-103: 214–229, 2016. It serves to familiarize the interested reader with the structure of the implementation, with the meaning of individual functions executed during the solution process, and to provide hints on how to adjust the code for individual needs. We would like to emphasize that the implementation is by no means optimal nor efficient. It serves merely to document the implementation of the examples employed in the cited paper.

Keywords: file system, compilation, implementation description, code adjustments

Contents

*Corresponding author.
Email address: rokosondrej@gmail.com (Ondřej Rokoš)

1. Introduction

In this work, we would like to guide the interested reader through the implementation that has been used to compute the results presented in (Rokoš et al., 2016, Section 5). The employed solution approach uses the Alternating Minimization (AM) procedure for two internal variables $\boldsymbol{r}$ and $\boldsymbol{z}_{\mathrm{p}}$, representing all atoms' positions in deformed configuration and plastic elongations of all bonds. Two summation rules, the exact rule from Beex et al. (2011), and the central from Beex et al. (2014), are used to approximate the reduced incremental energy Π^k_{red}. As outputs, the deformed configuration at the end of the evolution process and the energy evolution paths are plotted. Note that the external force vector, $\boldsymbol{f}_{\mathrm{ext}}$, is not present since both examples use only Dirichlet boundary conditions.

Source files can be accessed via

- MATLAB® Central
- GitHub repository

Please feel free to report any bugs or improvements on rokosondrej@gmail.com.

2. File Structure

Downloaded repository has the file structure depicted in Fig. 1.

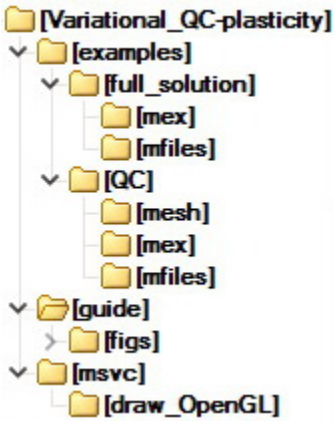

Figure 1: File system of downloaded repository.

Concerning the implementation itself, sub-folders `full_solution` and `QC`, both inside the `examples` folder, directly contain MATLAB m-files that serve to run one of the two examples. These are named as `RUN_*.m`. Other m-files, collected in `mfiles` sub-folders, are supporting and serve, e.g., to construct a mesh or to minimize the potential energy with

respect to $\boldsymbol{r}$. MATLAB mex functions, gathered inside `mex` sub-folders, are provided to improve the overall performance. In order to run any example, all included mex-files *must be* compiled first, cf. Section 3.1.

The `msvc` sub-folder comprises an optional post-processing tool that helps to view computed results (available only for pc). Its source code and Microsoft Visual Studio (MSVC) 2013 project files are contained in `draw_OpenGL` folder; compilation procedure is detailed in Section 3.2

3. Compilation

3.1. REQUIRED: Basic Compilation

In order to compile, a C++ compiler has to be present on the local machine. For the list of supported and compatible compilers refer here.

As a next step, installed compiler must be linked to MATLAB. This is done by executing `mex -setup C++` in MATLAB's command window, which prints available compilers: choosing one of them completes the linking process. Note that if a compiler is already linked, the message indicating its type and version will be printed. If an error occurs, the compiler has not been installed properly. In that case please follow the recommended procedures for the proper installation of particular compiler in your operating system (OS).

If a compiler is successfully installed, or was already present on the local machine, run `examples/full_solution/compile_mex.m` and `examples/QC/compile_mex_QC.m` in order to compile provided source files.

The implementation has been tested using Matlab 2016a and Microsoft Visual studio 2013 on pc (Windows), gcc 5.4.0 on unix, and Xcode 7.3.1 on mac.

3.2. OPTIONAL: Drawing Tool (for pc only)

Applies for MSVC only: first download freeglut, glew, and glui packages, unpack them and build solutions whenever necessary (this routine was verified for freeglut 3.0.0.1, glew 1.13.0, glui 2.35, and MSVC 2013).

In order to compile glui, open `glui/src/msvc/glui.sln` in MSVC. First change *Solution Configurations* to *Release* and set to x64 if necessary (*Configuration manager* → *Platform* → *New* → *x64*). In *Solution Explorer*, right-click on *_glui library* and go to *Properties* → *VC++ Directories*, where in *Include Directories* add `freeglut/include` directory. Subsequently, right-click on *_glui library* and *Build*.

Open `codes_plasticity/msvc/draw_OpenGL/draw_OpenGL.sln`. Perhaps, a conversion to a newer MSVC version will be required since `*.sln` file was created for MSVC 2013. In *Configuration manager* select *Win32* or *x64* for *Solution Platforms*, and choose *Release* in *Solution Configurations*. Go to *Solution Explorer*, right-click on *draw_OpenGL*, go to *Properties* → *VC++ Directories*, where in *Include Directories* add `freeglut/include`, `glew/include`, and `glui/src/include`. Press apply. In *Library Directories* add `freeglut/lib` (or `freeglut/lib/x64`), `glew/lib/Release/x*`, and `glui/src/msvc/lib`; choose always those libraries that match your OS, i.e. x86 or x64. Press apply and go to *Linker* → *Input* → *Additional Dependencies* and here enter `freeglut.lib` (press enter for a new line),

`glew32.lib`, and `glui32.lib`. Press *OK*, *Apply*, and *OK*. Now, the solution can be built: *Build* → *Build Solution*.

Having successfully built the solution, go to `codes_plasticity/msvc/draw_OpenGL/Release` (or to `codes_plasticity/msvc/draw_OpenGL/x64/Release`) and copy `draw_OpenGL.exe` to the folder where `RUN_*.m` files are situated. Further, it is necessary to copy also `freeglut.dll` file from `freeglut/bin` and `glew32.dll` from `glew/bin/Release` to this location, since `draw_OpenGL.exe` uses them. Again, match your OS type.

Before execution of `call_OpenGL.mex` and `draw_OpenGL.exe`, scripts `RUN_*.m` automatically test for all the required files. Therefore, until all libraries and executables are present, the results will not be displayed.

The entire post-processor implementation is contained in one source file, `draw_OpenGL.cpp`, described briefly in Section 4.

4. Description of the Implementation

Each file or external function used during the solution process is shortly described below in this section. Namely, variables which can be adjusted such as problem dimensions, solver tolerances, mesh generators, or various flags are specified; further details can be found as comments in source files.

For convenience, we recall the full-lattice version of the AM algorithm in Algorithm 1. Here, two auxiliary variables $\widetilde{\boldsymbol{r}}$ and $\widetilde{\boldsymbol{z}}_{\mathrm{p}}$ are introduced. Note that superscript symbols (i) and $[l]$ relate to Newton iterations in the case of (i) or (j), and to AM iterations in the case of $[l]$. The symbols (end) or [end] denote appropriate converged quantities at the end of the iteration process. The remaining notation is explained in (Rokoš et al., 2016, Section 2.2.1).

4.1. Description of `*.m` *files*

RUN_uniform_loading executes the full-lattice simulation for the uniform loading case, see (Rokoš et al., 2016, Section 5.1). First, MATLAB is reset to its initial configuration, memory is wiped, and working paths are initialized. Subsequently, the script loads user-defined parameters specifying geometry, material data, constructs `atoms` and `bonds` databases (cf. Section 4.2), and assembles boundary conditions. The AM algorithm subsequently minimizes the energy through iterative execution of *minimize_r* and *return_mapping* functions for each time step t_k, until convergence is reached. Finally, results are shown: deformed configuration, energy evolution paths, or the post-processing step through *call_OpenGL* function is executed, if available.

The following list of input constants allows to adjust the example: `dSizeX` and `dSizeY` describe lattice spacings along x- and y-axes, `SizeX` and `SizeY` describe one half of the rectangular domain Ω_0 along x- and y-axes, `RigidX` and `RigidY` describe one halves of the central inclusion along x- and y-axes. `PotentialI` $= [E_{\mathrm{I}}, H_{\mathrm{I}}, \sigma_{0,\mathrm{I}}, \rho_{\mathrm{I}}]$, `PotentialM` $= [E_{\mathrm{M}}, H_{\mathrm{M}}, \sigma_{0,\mathrm{M}}, \rho_{\mathrm{M}}]$ characterise material parameters of the inclusion and surrounding matrix. `TOL_am`, `TOL_r`, `TOL_z`, and `TOL_g` specify relative error tolerances for the AM algorithm, for

Algorithm 1: Alternating minimization algorithm.

Data : definition of energies, boundary conditions, and tolerances; initial conditions for internal variables are set to $\mathbf{0}$

Result: evolution of state variables $\boldsymbol{r}$, $\boldsymbol{z}_\mathrm{p}$, and $\boldsymbol{z}_\mathrm{c}$ as functions of time t_k, $k = 1, \ldots, n_T$

initialize $\boldsymbol{r}(t_0) := \boldsymbol{r}_0$, $\boldsymbol{z}_\mathrm{p}(t_0) := \mathbf{0}$, $\boldsymbol{z}_\mathrm{c}(t_0) := \mathbf{0}$
for $k := 1$ **to** n_T **do**
  initialize $\widetilde{\boldsymbol{r}}^{[0](\mathrm{end})} := \boldsymbol{r}(t_{k-1})$, $\widetilde{\boldsymbol{z}}_\mathrm{p}^{[0](\mathrm{end})} := \boldsymbol{z}_\mathrm{p}(t_{k-1})$, $\varepsilon_\mathrm{alt} := \mathrm{tol}_\mathrm{alt} + 1$
  % perform the AM procedure:
  set $l := 0$
  while $\varepsilon_\mathrm{alt} > \mathrm{tol}_\mathrm{alt}$ **do**
    initialize $\widetilde{\boldsymbol{r}}^{[l+1](0)} := \widetilde{\boldsymbol{r}}^{[l](\mathrm{end})}$, $\widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](0)} := \widetilde{\boldsymbol{z}}_\mathrm{p}^{[l](\mathrm{end})}$
    % minimize $\Pi^k_\mathrm{red}(\widehat{\boldsymbol{r}}, \widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](0)}; \boldsymbol{z}_\mathrm{p}(t_{k-1}), \boldsymbol{z}_\mathrm{c}(t_{k-1}))$ with respect to $\widehat{\boldsymbol{r}}$:
    set $i := 0$, initialize $\boldsymbol{f}_{\boldsymbol{r}}^{(0)} := -\partial\Pi(\boldsymbol{r})/\partial\boldsymbol{r}|_{\boldsymbol{r}=\widetilde{\boldsymbol{r}}^{[l+1](0)}}$
    while $||\boldsymbol{f}_{\boldsymbol{r}}^{(i)}||_2 > \mathrm{tol}_{\boldsymbol{r}}$ **do**
      $\boldsymbol{K}_{\boldsymbol{r}}^{(i)} := \partial^2\Pi(\boldsymbol{r})/\partial\boldsymbol{r}\partial\boldsymbol{r}|_{\boldsymbol{r}=\widetilde{\boldsymbol{r}}^{[l+1](i)}}$
      $\widetilde{\boldsymbol{r}}^{[l+1](i+1)} := \widetilde{\boldsymbol{r}}^{[l+1](i)} + (\boldsymbol{K}_{\boldsymbol{r}}^{(i)})^{-1}\boldsymbol{f}_{\boldsymbol{r}}^{(i)}$
      update $\boldsymbol{f}_{\boldsymbol{r}}^{(i+1)} := -\partial\Pi(\boldsymbol{r})/\partial\boldsymbol{r}|_{\boldsymbol{r}=\widetilde{\boldsymbol{r}}^{[l+1](i+1)}}$
      set $i := i + 1$
    end
    % minimize $\Pi^k_\mathrm{red}(\widetilde{\boldsymbol{r}}^{[l+1](\mathrm{end})}, \widehat{\boldsymbol{z}}_\mathrm{p}; \boldsymbol{z}_\mathrm{p}(t_{k-1}), \boldsymbol{z}_\mathrm{c}(t_{k-1}))$ with respect to $\widehat{\boldsymbol{z}}_\mathrm{p}$:
    set $j := 0$, initialize $\boldsymbol{f}_{\boldsymbol{z}}^{(0)} := -\partial\Pi(\boldsymbol{z}_\mathrm{p})/\partial\boldsymbol{z}_\mathrm{p}|_{\boldsymbol{z}_\mathrm{p}=\widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](0)}}$
    while $||\boldsymbol{f}_{\boldsymbol{z}}^{(j)}||_2 > \mathrm{tol}_{\boldsymbol{z}}$ **do**
      % Newton's method either for smoothing or return-mapping algorithm with non-linear hardening
      $\boldsymbol{K}_{\boldsymbol{z}}^{(j)} := \partial^2\Pi(\boldsymbol{z}_\mathrm{p})/\partial\boldsymbol{z}_\mathrm{p}\partial\boldsymbol{z}_\mathrm{p}|_{\boldsymbol{z}_\mathrm{p}=\widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](j)}}$
      $\widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](j+1)} := \widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](j)} + (\boldsymbol{K}_{\boldsymbol{z}}^{(j)})^{-1}\boldsymbol{f}_{\boldsymbol{z}}^{(j)}$
      update $\boldsymbol{f}_{\boldsymbol{z}}^{(j+1)} := -\partial\Pi(\boldsymbol{z}_\mathrm{p})/\partial\boldsymbol{z}_\mathrm{p}|_{\boldsymbol{z}_\mathrm{p}=\widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](j+1)}}$
      set $j := j + 1$
    end
    $\varepsilon_\mathrm{alt} := ||\widetilde{\boldsymbol{z}}_\mathrm{p}^{[l+1](\mathrm{end})} - \widetilde{\boldsymbol{z}}_\mathrm{p}^{[l](\mathrm{end})}||_2$
    set $l := l + 1$
  end
  $\boldsymbol{r}(t_k) := \widetilde{\boldsymbol{r}}^{[\mathrm{end}](\mathrm{end})}$, $\boldsymbol{z}_\mathrm{p}(t_k) := \widetilde{\boldsymbol{z}}_\mathrm{p}^{[\mathrm{end}](\mathrm{end})}$,
  $\boldsymbol{z}_\mathrm{c}(t_k) := \boldsymbol{z}_\mathrm{c}(t_{k-1}) + |\boldsymbol{z}_\mathrm{p}(t_k) - \boldsymbol{z}_\mathrm{p}(t_{k-1})|$
end

the Newton's method with respect to $\boldsymbol{r}$, for the Newton's method used during the return-mapping algorithm when $\boldsymbol{z}_{\mathrm{p}}$ variable is being computed, and the geometric tolerance (a distance that is smaller than `TOL_g` is treated as zero). Dirichlet boundary conditions, namely x-displacement magnitude of Γ_2, is specified in `u_D`. Pseudo-time profile and the number of time increments n_T can be adjusted in `Time` variable. Concerning basic outputs, the `scale` variable prescribes the magnification factor of the displacements for the final deformed configuration. Above, Young's modulus E, hardening modulus H, initial yield stress σ_0, and hardening exponent ρ specify particular physical parameters of the lattice.

RUN_pure_bending provides the solution of the pure bending example presented in (Rokoš et al., 2016, Section 5.2) for the full-lattice formulation. Structure of the code is identical to the one in `RUN_uniform_loading.m` file; several marginal differences in formulation of boundary conditions occur, however. In particular, `RigidY` describes the entire size of the bottom-edge inclusion along y-axis instead of one half; instead of `u_D`, the parameter `THETA` specifies target deformation.

RUN_indentation solves the example presented in (Rokoš et al., 2016, Section 5.3) for the full-lattice formulation. Structure of the code is identical to the two previous ones, `RUN_uniform_loading.m` and `RUN_pure_bending.m`. The only differences are in the formulation of the boundary conditions, used minimization procedure, and in presence of two new parameters specifying the indenter. Regarding boundary condition, all three parts of the boundary $\Gamma_{1,2,4}$ are fixed in horizontal as well as vertical direction, whereas Γ_3 is left free. Here, contact with the indenter is assumed. The position of the indenter is specified through its centre `C` which depends on `Time`. In order to minimize Π^k_{red} with respect to kinematic variables, function *minimize_r_I* or *minimize_r_ISQP* is called, which incorporates inequality constraints through Lagrange multipliers and active-set strategy. The new parameters are string `indenter`, which specifies indenter's profile ("square" or "circular"), and `Rind`, which specifies its size (half-side of the square or radius of the circular indenter). Because the specimen is homogeneous, only one `Potential` $= [E, H, \sigma_0, \rho]$ characterizes material properties.

grad_hess provides the full vector $\boldsymbol{r}$ denoted as `r2` (which is reconstructed from the current iteration variable $\boldsymbol{x}$), and gradient and Hessian of the reduced incremental energy Π^k_{red} for the full-lattice system with respect to $\boldsymbol{r}$. This function effectively reduces to the execution of the *build_grad_r* and *build_hess_r* procedures followed by the application of prescribed Dirichlet boundary conditions.

minimize_r executes the minimization step (AMa) with respect to $\boldsymbol{r}$ at a fixed time step t_k and AM iteration l. First, *grad_hess* function is called to provide the gradient, Hessian, and current-iteration configuration $\boldsymbol{r}$. Then, tying conditions are introduced and the saddle point problem assembled and solved. Convergence of the Newton's algorithm is

achieved when relative error is smaller than `TOL_r`. After convergence, full vector $\boldsymbol{r}^{[l]}(t_k)$, denoted as `r2`, is returned together with the number of Newton iterations `Niter`.

minimize_r_I minimizes the incremental energy with respect to kinematic variable $\boldsymbol{r}$ in the AM algorithm, cf. *minimize_r*. Contrary to *minimize_r*, inequality constraints are incorporated through primal-dual formulation and active-set strategy.

Atoms that may come into contact with the indenter (and are therefore tested for penetration) are stored in `IDGamma_3` array. Indices of those from `IDGamma_3` that actually are in the contact are listed in `IDActive`. For each atom in `IDActive`, associated inequality constraint is enforced as equality constraint (linear for square and nonlinear for circular indenter). Subsequently, Newton's algorithm is iterated until convergence. Upon convergence, active set is updated and if necessary, the system is relaxed again until both, `IDActive` set and Newton's algorithm converge. Function *minimize_r_I* returns amongst others `IDActive` array that is used as the initial estimate for the next step or AM iteration.

minimize_r_ISQP works as an alternative to *minimize_r_I*. In order to deal with inequality constraints, this function employs the Sequential Qudaratic Programming (SQP) algorithm along with active set strategy. Further details on SQP and active set can be found e.g. in Fletcher (1987); Bonnans et al. (2006); Nocedal and Wright (2006).

RUN_uniform_loading_QC solves the uniform loading example using the QC approach. Apart from the procedures contained in `RUN_uniform_loading.m`, two additional steps have to be accomplished. Firstly, for the interpolation, the domain Ω_0 has to be triangulated. Here, four options are offered: (1) load one of the meshes presented in (Roko$\check{\text{s}}$ et al., 2016, Fig. 5) and provided in the `mesh` folder; (2) build a regular mesh using right-triangulated irregular networks (RTIN) and longest-edge-propagation-path (LEPP) refinement algorithm after Rivara (1997); (3) build a mesh using $\mathcal{T}3\mathcal{D}$ generator through *mesh_T3d_QC*; (4) call the MATLAB mesh generator *initmesh* through *mesh_MATLAB_QC*. Secondly, for the summation rule, the database of sampling atoms is specified through *sort_sampling_atoms_QC* function. Boundary and tying conditions for repatoms are finally provided by `build_phi_u_QC.m`.

In comparison with the full solution *RUN_uniform_loading*, couple of additional variables have to be specified. `SumRule` characterises summation rule to be used: 0 for the exact summation rule, 1 for the central summation rule (both after Beex *et al.*), and 2 for the full summation rule. `MeshType` specifies domain triangulation to be used: 0 for using one of the provided meshes, 1 for using RTIN mesh, 2 for using the $\mathcal{T}3\mathcal{D}$ generator, and 3 for using the MATLAB mesh generator. Finally, `FineX` and `FineY` describe half sizes of the fully-resolved mesh region along x- and y-axes.

RUN_pure_bending_QC solves the pure bending example using the QC approach; for completeness, refer also to *RUN_uniform_loading_QC* and *RUN_uniform_loading*. Four options for the interpolation step are provided again: use stored meshes described in (Roko$\check{\text{s}}$ et al., 2016, Fig. 9), RTIN, $\mathcal{T}3\mathcal{D}$, or MATLAB.

Contrary to *RUN_uniform_loading_QC* function, `RigidY` now describes the entire size of the bottom-edge inclusion along the y-axis instead of one half; analogously for `FineY`. Instead of `u_D`, which specifies the Dirichlet boundary conditions, angle `THETA` is prescribed.

RUN_indentation_QC provides results for the indentation example using QC approach. Functions *minimize_r_I_QC* or *minimize_r_ISQP_QC* minimize the incremental energy with respect to kinematic variable, cf. *RUN_indentation*.

grad_hess_QC provides current-iteration vector $\boldsymbol{r}_{\text{rep}}$, denoted as `r2`, together with the gradient and Hessian of the approximate incremental energy $\widehat{\Pi}^k_{\text{red}}$ with respect to the reduced variable $\boldsymbol{r}_{\text{rep}}$. In the function call, first the full vector $\boldsymbol{r}$ is reconstructed through the interpolation matrix $\boldsymbol{\Phi}$. Then, using chosen summation rule, full-dimensional approximate gradient and Hessian are computed through *build_grad_r_QC* and *build_hess_r_QC* mex-functions. Finally, the reduction step is carried out through the interpolation matrix $\boldsymbol{\Phi}$ and the Dirichlet boundary conditions are prescribed.

minimize_r_QC minimizes the approximate reduced incremental energy $\widehat{\Pi}^k_{\text{red}}$ with respect to $\boldsymbol{r}_{\text{rep}}$. Iteratively, *grad_hess_QC* function is called to provide the reduced gradient and Hessian, followed by incorporation of the tying conditions for $\boldsymbol{r}_{\text{rep}}$ to yield the saddle point problem, cf. also *minimize_r*. Converged vectors $\boldsymbol{r}_{\text{rep}}$ and $\boldsymbol{r}$, denoted as `r2` and `r`, and the number of Newton iterations `Niter` are returned.

minimize_r_I_QC minimizes the approximate incremental energy with respect to kinematic variable, see also *minimize_r_QC*. This function is extended to reflect also inequality constraints due to indenter. Used strategy is the same as in *minimize_r_I*, applied only to repatoms instead of to all atoms.

minimize_r_ISQP_QC is an alternative to *minimize_r_QC*. In analogy to *minimize_r_ISQP*, this function employs SQP algorithm with active set strategy.

mesh_MATLAB_QC depending on the example type (uniform loading or pure bending), this function assembles required inputs to MATLAB *initmesh* function, and subsequently generates the mesh. Since all the mesh nodes have to spatially conform to the underlying lattice, node coordinates are rounded with respect to `dSizeX` and `dSizeY` and the resulting mesh is checked for hanging nodes in `RUN_*.m`. Function returns an array of final triangles `t` and an array of node points `p`. For the definition of `t` and `p` see MATLAB help for the keyword *initmesh*. Note that as an alternative to this approach, one could use Delaunay triangulation. In that case, repatoms have to be chosen carefully.

`HMax` variable limits longest edges of triangles.

mesh_T3d_QC depending on the example type, this function assembles an input file `T3d.in` for the $\mathcal{T}3\mathcal{D}$ mesh generator. Subsequently, `T3d.exe` routine with several options is called; for a closer description of this program and its download please refer here. $\mathcal{T}3\mathcal{D}$ generates an output file called `T3d.out`, which is parsed and converted to `p` and `t` arrays. Then, node coordinates are rounded with respect to `dSizeX` and `dSizeY`.

mesh_RTIN_QC takes as inputs constants specifying the sizes of the rectangular domain and finely refined region, and returns the triangulations represented by point and triangle matrices `p` and `t`. Initially, a coarse regular mesh is constructed, which is iteratively (by calling *regular_refine*) refined until all required points of the fully-resolved region become repatoms.

regular_refine refines current triangulation according to specified set of triangles marked for refinement, `refine_triangles`. The fully refined triangles are first excluded from the list, *refine_mesh* algorithm subsequently performs actual mesh refinement. Matrices `p` and `t`, that represent refined triangulation, are returned.

4.2. Description of `*.cpp` *mex-files*

build_atoms serves first to build the database of atoms, i.e. to provide a system of structures

`atoms(`α`).R(2)`
`.NeighbourList(n)`
`.BondList(n)`

for $\alpha = 1, \ldots, n_{\text{ato}}$, where `R` $= [r_{0,x}, r_{0,y}]$, and `NeighbourList` is an `n`-dimensional vector, storing IDs of atoms contained in the set B_α; note that `n` $= \#B_\alpha$. Similarly, `BondList`, also an array of the length $\#B_\alpha$, is allocated to store IDs of bonds connecting an atom α with its nearest neighbours stored in B_α, yet left empty. Bond IDs are supplied later by *build_bonds* function call. Note that the array `R0` $= [\boldsymbol{r}_0^1, \ldots, \boldsymbol{r}_0^{n_{\text{ato}}}]^\mathsf{T}$, `R0`$(2\alpha-1, 2\alpha) = [r_{0,x}^\alpha, r_{0,y}^\alpha]^\mathsf{T}$, i.e. the vector of all atoms' initial positions, can be constructed as `R0` $=$ `[atoms(:).R]`$'$.

As an input, the array `[SizeX, SizeY, dSizeX, dSizeY]` storing dimensions of the problem has to be supplied. It represents a rectangular domain $\Omega_0 =$ `[-SizeX, SizeX]` $\times$ `[-SizeY, SizeY]` with atom spacings `dSizeX` along x- and `dSizeY` along y-axes.

build_bonds provides the database of bonds, i.e.

`bonds(`m`).Atoms(2)`
`.Potential(4)`

for $m = 1, \ldots, n_{\text{bon}}$. Each bond connects atoms α and β stored in `Atoms` $= [\alpha, \beta]$, and has material parameters specified in `Potential` $= [E^{\alpha\beta}, H^{\alpha\beta}, \sigma_0^{\alpha\beta}, \rho^{\alpha\beta}]$. Possible reduction of cross-sectional bond area is accomplished by scaling E and σ_0. This function provides also `atoms`(α)`.BondList` for each α, discussed in *build_atoms* function description. Although in the context of homogeneous lattice structures this kind of representation is far from efficient, it is convenient for the representation of inhomogeneous or random structures.

As an input, following variables have to be provided: `atoms`, [`SizeX`, `SizeY`, `dSizeX`, `dSizeY`, `RigidX`, `RigidY`], `PotentialI`, `PotentialM`, `flag`, and `TOL_g`. For the definition of `atoms` and [`SizeX`, `SizeY`, `dSizeX`, `dSizeY`] refer to *build_atoms*. `RigidX` and `RigidY` describe the rigid inclusion's dimensions having the material parameters stored in `PotentialI`. `PotentialM` specifies the material parameters assigned to the surrounding matrix; both inputs are structured as `PotentialI` $= [E_{\mathrm{I}}, H_{\mathrm{I}}, \sigma_{0,\mathrm{I}}, \rho_{\mathrm{I}}]$, `PotentialM` $= [E_{\mathrm{M}}, H_{\mathrm{M}}, \sigma_{0,\mathrm{M}}, \rho_{\mathrm{M}}]$. Variable `flag` chooses the location of the inclusion: 0 for the central inclusion, 1 for the bottom-edge inclusion. The last input parameter is `TOL_g`, set as default to 10^{-10}. When algorithm decides which atoms belong to the inclusion $\mathscr{I}$, it first computes the ℓ_1-distance $\mathrm{dist}(\boldsymbol{r}_0^\alpha, \mathscr{I})$ between the position vector $\boldsymbol{r}_0^\alpha$ and the inclusion $\mathscr{I}$. Therefore, if $\mathrm{dist}(\boldsymbol{r}_0^\alpha, \mathscr{I}) <$ `TOL_g`, then $\alpha \in \mathscr{I}$. Bond $\alpha\beta \in \mathscr{I}$ if $\alpha \in \mathscr{I}$ and $\beta \in \mathscr{I}$ at the same time.

build_grad_r returns a vector `f_r` storing the first derivatives of the incremental energy with respect to $\boldsymbol{r}$, i.e. `f_r` $= \partial\Pi_{\mathrm{red}}^k/\partial\boldsymbol{r} = \partial\mathcal{V}^{\mathrm{int}}/\partial\boldsymbol{r}$. As inputs, `atoms`, `bonds`, `r`, and `z` variables have to be provided. Here, `atoms` is the database of all atoms, `bonds` the database of all bonds, `r` $= [\boldsymbol{r}^1, \ldots, \boldsymbol{r}^{n_{\mathrm{ato}}}]^{\mathsf{T}}$, `r`$(2\alpha-1, 2\alpha) = [r_x^\alpha, r_y^\alpha]^{\mathsf{T}}$ stores current positions of all atoms $\alpha = 1, \ldots, n_{\mathrm{ato}}$ (and represents the counterpart to $\widetilde{\boldsymbol{r}}$ variable in Algorithm 1), whereas `z` $= [z_p^1, \ldots, z_p^{n_{\mathrm{bon}}}]^{\mathsf{T}}$ stores plastic slips for all bonds $\alpha\beta$, being the counterpart to $\widetilde{\boldsymbol{z}}_{\mathrm{p}}$.

build_hess_r returns three two-dimensional full matrices `I`, `J`, and `S` storing row indices, column indices, and values of the Hessian. In other words, this function provides the so-called COO sparse matrix representation of the Hessian. Employing MATLAB's function `K_r` $=$ *sparse*(`I(:)`, `J(:)`, `K(:)`, `n`, `n`) for vectorised matrices `I`, `J`, and `K`, a sparse `n` $\times$ `n` matrix representing the global Hessian of the reduced incremental energy, Π_{red}^k, with respect to $\boldsymbol{r}$ is computed, i.e. `K_r` $= \partial^2\Pi_{\mathrm{red}}^k/\partial\boldsymbol{r}\partial\boldsymbol{r} = \partial^2\mathcal{V}^{\mathrm{int}}/\partial\boldsymbol{r}\partial\boldsymbol{r}$; above, `n` $= 2n_{\mathrm{ato}}$. For further details on assembling stiffness matrices in MATLAB see Japhet et al. (2013). As an input, `atoms` and `bonds` databases have to be provided as well as the vector of current atoms' positions `r` and plastic deformations `z`. See *build_grad_r* paragraph for the definitions of `r` and `z`.

return_mapping performs the minimization step (AMb), i.e. minimizes the reduced incremental energy Π_{red}^k with respect to internal variables $\boldsymbol{z}_{\mathrm{p}}$ at a fixed time step and AM iteration, t_k and l. This function returns a vector of plastic deformations, denoted as `z2`. Rewriting Π_{red}^k in the bond-wise form, this amounts to independent minimization of each non-smooth bond-energy $\widetilde{\pi}_m^k(z_{\mathrm{p}}^m)$ with respect to z_{p}^m for $m = 1, \ldots, n_{\mathrm{bon}}$. As inputs, `atoms` and `bonds` databases have to be supplied. Further, updated deformation vector `r` as well as previous-time-step internal variables `Z(:,k-1)` $= [z_{\mathrm{p}}^1(t_{k-1}), \ldots, z_{\mathrm{p}}^{n_{\mathrm{bon}}}(t_{k-1})]^{\mathsf{T}}$ and `K(:,k-1)` $= [z_{\mathrm{c}}^1(t_{k-1}), \ldots, z_{\mathrm{c}}^{n_{\mathrm{bon}}}(t_{k-1})]^{\mathsf{T}}$ are required. Finally, iteration tolerance `TOL_z` specifies relative tolerance of the Newton's algorithm for non-linear hardening rule. For detailed description of the return-mapping algorithm refer e.g. to (Simo and Hughes, 2000, Section 1.4.2).

`MAXITER` constant determines the maximum number of Newton iterations that can be

taken when returning back to yield surface for non-linear hardening rule and a single bond. Default value is set to 100.

build_en returns the elastic part $\mathcal{E}(t_k)$ of the incremental energy $\Pi^k = \mathcal{E}(t_k)+\mathcal{D}(t_k, t_{k-1})$. As inputs, `atoms` and `bonds` databases, atom positions `R(:,k)` $= \boldsymbol{r}(t_k)$, plastic elongations `Z(:,k)` $= \boldsymbol{z}_{\mathrm{p}}(t_k)$, and cumulated plastic elongations `K(:,k)` $= \boldsymbol{z}_{\mathrm{c}}(t_k)$ in current time step t_k have to be provided. For a closer description of `Z` and `K` refer to *return_mapping* function.

build_diss returns the dissipation part $\mathcal{D}$ of the incremental energy Π^k, i.e. the dissipation distance from the previous to the current time step. Apart from `atoms` and `bonds` databases, variables `Z(:,k-1)` and `Z(:,k)` are required. For their definitions refer to *return_mapping*.

call_OpenGL firstly maps all the resulting data required for the postprocessing step from MATLAB to the OS' memory, i.e. `atoms`, `samplingatoms`, `repatoms`, `bonds`, `sampbondsID`, `triangles`, `R`, `Z`, `K`, `Time`, `SizeX`, and `SizeY`. Subsequently, the secondary process `draw_OpenGL.exe` that draws the results is called.

sort_atoms_QC serves to provide five outputs. First, the database of triangles, i.e. an array of structures

```
triangles(k).P1(2)
            .P2(2)
            .P3(2)
            .T(2)
            .IntAtoms(nI)
            .EdgeAtoms(nE)
            .VertexAtoms(nV)
            .NeighTriangles(nN)
```

for `k` $= 1, \ldots,$ `Nt`, where `Nt` is the number of triangles, is provided. Above, `P1` – `P3` store coordinates $[p_x, p_y]^{\mathsf{T}}$ of the three vertex atoms or nodes of the `k`-th triangle, `T` $= [c_x^{\mathtt{k}}, c_y^{\mathtt{k}}]^{\mathsf{T}}$ stores coordinates of the centroid (or incenter, circumcenter) for the `k`-th triangle. `Int-`, `Edge-`, and `Vertex-atoms` store atom IDs of particular atom types; `VertexAtoms` vector is sorted in correspondence to `P1`, `P2`, and `P3`. `NeighTriangles` collects IDs of all the neighbouring triangles. Second, this function provides a vector `repatoms` $= N_{\mathrm{rep}}^{\mathrm{ato}}$, that stores all repatoms' IDs. Third, column arrays `I`, `J`, and `S` are provided, which store the COO data for $\boldsymbol{\Phi}$ matrix. Using MATLAB's function `Phi` $= accumarray([\mathtt{I}, \mathtt{J}], \mathtt{S}, [2\mathtt{n}, 2\mathtt{n\_r}], @\max, \mathrm{true})$, a sparse matrix $\boldsymbol{\Phi}$ of size $2\mathtt{n} \times 2\mathtt{n\_r}$, where `n` $= n_{\mathrm{ato}}$ and `n_r` $= n_{\mathrm{rep}}$, is computed. As inputs, `p` and `t` matrices (cf. *mesh_MATLAB_QC*) along with `atoms` database and `TOL_g` (cf. *RUN_uniform_loading*) scalar have to be provided.

Before compilation, user can specify through `FLAG` which coordinates will be stored in `T`: 1 for centroid, 2 for incenter, or 3 for circumcenter. Default value is set to 2.

sort_sampling_atoms_QC provides `samplingatoms` database, where each entry has the following form

```
samplingatoms(l).ID
               .w
               .List(n)
```

for $\mathtt{l} = 1, \ldots, n_{\mathrm{sam}}^{\mathrm{ato}}$, where `ID`s are sampling atoms' IDs in `atoms` database, `w`s their weight factors, and the variable `List` stores IDs of atoms that are represented by `l`-th sampling atom. Required inputs consist of six variables: `atoms` database, `triangles` database, `SumRule` switch specifying the summation rule to be used, [`SizeX`, `SizeY`] vector describing the domain geometry, `SW` specifying the example being solved (0 for uniform loading and 1 for pure bending), and `TOL_g`. A distance $\bullet$ for which $|\bullet| <$ `TOL_g` holds is considered as zero; here it serves to locate atoms lying at $\partial\Omega_0$. Let us recall `FLAG` switch from the function *sort_atoms_QC*, which specifies whether centroid, 1, incenter, 2, or circumcenter, 3, is stored in `T`. In *sort_sampling_atoms_QC*, `T` is used as the reference point for choosing the central sampling atom. Atom that is closest to `T` is chosen. Note, however, that an atom with a higher number of neighbours inside the triangle is chosen preferably. Therefore, the number of neighbours outweighs the distance criterion.

build_grad_r_QC serves to compute the gradient of the reduced approximate energy $\widehat{\Pi}^k_{\mathrm{red}}$, cf. *build_grad_r*; an additional input `samplingatoms` in contrast to the full gradient has to be provided.

build_hess_r_QC computes the Hessian of the reduced approximate energy $\widehat{\Pi}^k_{\mathrm{red}}$, cf. *build_hess_r*; an additional input `samplingatoms` compared to the full Hessian has to be provided.

return_mapping_QC performs the minimization step (AMb) of $\widehat{\Pi}^k_{\mathrm{red}}$ with respect to $\boldsymbol{z}_{\mathrm{p,sam}}$ at a fixed time step and AM iteration (t_k and l) through the return-mapping algorithm. Function returns $\boldsymbol{z}_{\mathrm{p,sam}}$, stored in fully-dimensional vector `z2`; `samplingbondsID` vector in addition to inputs described in *return_mapping* is required.

build_en_QC provides the elastic part of the approximate incremental energy $\widehat{\Pi}^k$ using a summation rule. Besides the inputs specified in *build_en*, `samplingatoms` database has to be provided.

build_diss_QC computes dissipation distance part of the approximate incremental energy $\widehat{\Pi}^k$, cf. also *build_diss*. This function requires also `samplingatoms` database in addition to `Z(:,k-1)` and `Z(:,k)`.

refine_mesh performs the Backward-Longest-Edge-Bisection algorithm for a mesh specified by the `p` and `t` matrices and for triangles marked for refinement stored in the vector

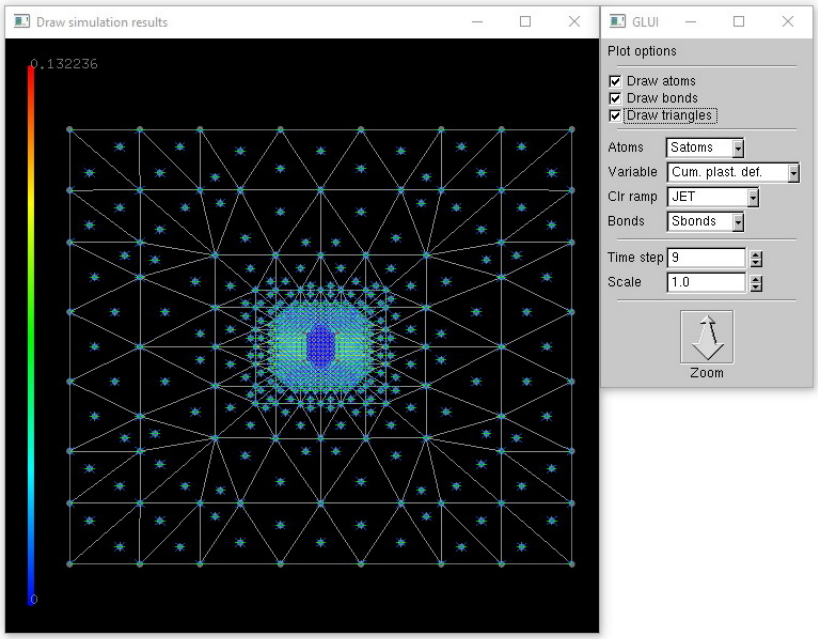

Figure 2: A print screen of the drawing tool.

`refine_triangles`; for additional details see Rivara (1997). Further inputs are switch variable `SW` and the geometric tolerance `TOL_g`. The geometric tolerance serves the usual purpose, i.e. the distance for which $|\bullet| <$ `TOL_g` is treated as zero, whereas the switch decides if the permutation of the `t` matrix will be verified. If `SW` $= 0$, no permutation takes place. If `SW` $= 1$ indices of each triangle `t(:,i)` are checked and permuted such that the first two indices correspond to the longest edge of the triangle. In its fourth row, matrix `t` contains IDs of parent elements from the previous mesh; this ID is negative if the triangle was not refined and positive otherwise.

Before compilation, user can specify `NLEPP` and `MULT` constants. `NLEPP` is the assumed length of the longest-edge-propagation path for the allocation of numerical arrays (default value is 100, though on average achieves value 7). Constant `MULT` serves to allocate output matrices (`p` and `t`). Namely, it tries to predict the ratio of the lengths of inputs to outputs (default value is 10).

4.3. Description of `*.cpp` *files*

draw_OpenGL closely communicates with *call_OpenGL* (it is its secondary process). First, data shared by *call_OpenGL* are assigned and buffered. Subsequently, glut frame with a simple glui menu is initialized and all user's commands are executed. After closing the program, buffers are unmapped such that memory allocated in *call_OpenGL* can be released. For further details see e.g. Hill and Kelly (2006). Glui is properly described in it's package manual.

Several options can be set during the program run, cf. Fig. 2. User can draw *atoms*, *bonds*, or *triangles* by marking appropriate checkboxes. Listbox for *Atoms* allows to draw

All atoms, *Satoms* (sampling atoms), or *Repatoms* (representative atoms). Internal variable that is being plotted is specified in *Variable* listbox, and has two items: *Plastic deformation* for $\boldsymbol{z}_{\mathrm{p}}$, and *Cum. plast. def.* for $\boldsymbol{z}_{\mathrm{c}}$. Values of $\boldsymbol{z}_{\mathrm{p}}$ and $\boldsymbol{z}_{\mathrm{c}}$ are depicted in four colour schemes specified by listbox *Clr ramp*: *RGB bipolar*, *Thermal*, *JET*, and *RWB*. The set of bonds that will be plotted can be chosen in *Bonds*: *All bonds* or *Sbonds* (sampling bonds). *Time step* spinner serves to choose for which time step k of t_k the results will be presented. *Scale* spinner specifies the deformation magnitude ranging from 0 to 100. Finally, drag-icon *Zoom* serves to zoom in or out of the plot. The entire scene can be translated in drag-and-pull manner pressing the left-mouse-button at the same time. Hitting "+" or "−" key will increase or decrease k of the time step t_k by one.

Acknowledgements

The financial support for this work from the Czech Science Foundation (GAČR) under project No. 14-00420S is gratefully acknowledged.

References

L. Beex, R. Peerlings, and M. Geers. Central summation in the quasicontinuum method. *Journal of the Mechanics and Physics of Solids*, 70(0):242 – 261, 2014. ISSN 0022-5096. doi: http://dx.doi.org/10.1016/j.jmps.2014.05.019. URL http://www.sciencedirect.com/science/article/pii/S0022509614001100.

L. A. A. Beex, R. H. J. Peerlings, and M. G. D. Geers. A quasicontinuum methodology for multiscale analyses of discrete microstructural models. *International Journal for Numerical Methods in Engineering*, 87(7):701–718, 2011. ISSN 1097-0207. doi: 10.1002/nme.3134. URL http://dx.doi.org/10.1002/nme.3134.

J. F. Bonnans, J. C. Gilbert, C. Lemaréchal, and C. A. Sagastizábal. *Numerical Optimization: Theoretical and Practical Aspects (Universitext)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 354035445X. doi: 10.1007/978-3-540-35447-5.

R. Fletcher. *Practical Methods of Optimization; (2Nd Ed.)*. Wiley-Interscience, New York, NY, USA, 1987. ISBN 0-471-91547-5.

F. Hill and S. M. Kelly. *Computer Graphics using OpenGL*. Prentice Hall, 2006.

C. Japhet, F. Cuvelier, and G. Scarella. An efficient way to perform the assembly of finite element matrices in Matlab and Octave. Feb. 2013. URL https://hal.archives-ouvertes.fr/hal-00785101.

J. Nocedal and S. J. Wright. *Numerical optimization*. Springer Series in Operations Research and Financial Engineering. Springer, Berlin, 2006. ISBN 978-0387-30303-1. URL http://opac.inria.fr/record=b1120179. NEOS guide http://www-fp.mcs.anl.gov/otc/Guide/.

M.-C. Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *International Journal for Numerical Methods in Engineering*, 40(18):3313–3324, 1997. ISSN 1097-0207. doi: 10.1002/(SICI)1097-0207(19970930)40:18⟨3313::AID-NME214⟩3.0.CO;2-\#.

O. Rokoš, L. A. A. Beex, J. Zeman, and R. H. J. Peerlings. A variational formulation of dissipative quasicontinuum methods. *International Journal of Solids and Structures*, 102–103: 214 – 229, 2016. ISSN 0020-7683. doi: http://dx.doi.org/10.1016/j.ijsolstr.2016.10.003. URL http://www.sciencedirect.com/science/article/pii/S0020768316302943.

J. Simo and T. Hughes. *Computational Inelasticity*. Interdisciplinary Applied Mathematics. Springer New York, 2000. ISBN 9780387975207. URL http://www.springer.com/us/book/9780387975207.